An Object Model of Direction And Its Implications*

Shashi Shekhar, Xuan Liu, Sanjay Chawla
Computer Science Department, University of Minnesota
EE/CS 4-192, 200 Union St. SE., Minneapolis, MN 55455
telephone: (612)624-8307
$[shekhar|xliu]@cs.umn.edu$
http://www.cs.umn.edu/research/shashi-group

Abstract

Direction is an important spatial concept that is used in many fields such as geographic information systems(GIS) and image interpretation. It is also frequently used as a selection condition in spatial queries. Previous work has modeled direction as a relational predicate between spatial objects. Conversely, in this paper, we model direction as a new kind of spatial object using the concepts of vectors, points and angles. The basic approach is to model direction as a unit vector. This novel view of direction has several obvious advantages: Being modeled as a spatial object, a direction object can have its own attributes and operation set. Secondly, new spatial data types such as oriented spatial objects and open spatial objects can be defined at the abstract object level. Finally, the object view of direction makes direction reasoning easy and also reduces the need for a large number of inference rules. These features are important in spatial query processing and optimization. The applicability of the direction model is demonstrated by geographic query examples.

Keywords: Direction, Orientation, Open object, Oriented spatial object

[1]This work is sponsored in part by the Army High Performance Computing Research Center under the auspices of the Department of the Army, Army Research Laboratory cooperative agreement number DAAH04-95-2-0003/contract number DAAH04-95-C-0008, the content of which does not necessarily reflect the position or the policy of the government, and no official endorsement should be inferred. This work was also supported in part by NSF grant #9631539

1 Introduction

1.1 Modeling Direction

Direction is a common spatial concept that is used everywhere in daily life. When people communicate about a geographic space, for example, giving route descriptions, direction is necessary to convey the information. Direction is also frequently used as a selection condition in spatial queries[11] or used for similarity accessing in image databases[18]. Example queries used in army battlefield visualization[10] are "Is there anything *over* the ridge?," "List the swamps in *front* of the tank."and "Let's move to the *north* of the tree." The first example refers to a viewer-based orientation, the second can be defined on either the intrinsic orientation of the tank(object-based) or of a viewer, and the third example refers to the absolute direction with respect to the tree. In order to process queries involving direction constraints in the selection criteria, a spatial database system should provide a way for users to formalize directions.

The common means of handling direction is to model direction as a spatial relationship between objects[4, 20, 8, 26, 3, 16, 9, 6, 22, 15]. In this paper, we formalize direction from a different perspective: as a spatial object. The basic approach is to model direction as a unit vector and orientation as a set of directions. As a spatial object, direction can have its own attributes, and its own operators, and a richer set of predicates and operators on direction and orientation can be defined. Second, new spatial data types such as oriented spatial objects and unbounded spatial objects can be easily defined in the abstract object level. The object view of direction also makes it easy to perform direction reasoning by using only simple vector algebra, which also reduces the large number of inference rules commonly needed. This is useful in the processing and optimization of spatial queries that contain direction constraints. We model the direction for three frames of reference: absolute, object and viewer-based orientation

We will not discuss the philosophical issue of whether directions are object/entities or not in the strictest sense. Absolute directions such as North, South, East, or West can be defined using a coordinate system w.r.t. the earth, and thus these can be considered to be objects in their own right. The object view for relative directions (e.g. left, front) needs further thought, because these directions are defined with respect to coordinate systems attached to other objects. The issue needs to be explored further.

1.2 Related work and our contributions

The research work on direction modeling has been carried out in several areas such as geographic information systems and image analysis. Most of the studies is on how to capture the semantics of direction relations, and further, how to do spatial reasoning on the direction [3, 4, 6]. There are two major direction reference frames used to model direction in 2D space: the cone-based model[16], and the projection-based model[4, 6]. Frank[1] compared these two models and found the projection-based reference frame to be better in many aspects. The most common way to model directions between extended objects is through the object's Minimum Bounding Rectangle(MBR), where direction relations are obtained by applying Allen's [2] interval relations along the x and y axis, in which case, 169 different relations[3] can be distinguished. Some work based on MBR has been proposed on picture indexing in pictorial databases[18, 26], and some work aligns each boundary box to the object's major axis[14], which makes it possible to satisfy different reference frames[9]. On the other hand, Freska [5] proposed an alternative method: semi-intervals to formalize the one-dimensional temporal relation based on incomplete knowledge of the object. Goyal and Egenhofer [8] introduced a Direction-Relation Matrix to represent cardinal directions. Based on the projection-based frame, it partitions the space around the reference object and records into which direction tiles a target object falls. But this model still has limitations in the modeling of line objects, and it is limited to 2D space. Little work has been done on directions in 3D space [7].

The previous work modeled direction as a spatial relationship between spatial objects. This seems to be a natural mapping of the directional relationship that is used in geographic space. But this modeling method has some limitations. Operations on direction are limited. Oriented(directed) objects and unbounded objects cannot be represented in the spatial data model.

This paper models direction as a spatial object: a unit vector. This novel view of direction has several advantages over the binary boolean relationship view. Being modeled as a spatial object, a direction object can have its own attributes and operation set. The implementation of operators can use vector algebra, making a richer set of predicates and operators on direction feasible. Secondly, new spatial data types such as oriented objects and unbounded objects can be defined at the abstract object level. The object view of direction also makes directional reasoning easy. Basic vector algebra is sufficient for inferencing new directions, and no special or new qualitative rules are needed. This reduces the complexity of directional reasoning, and involving appropriate quantitative information is useful in spatial query processing and optimization in spatial databases.

More discussion on direction objects, formal proofs, and their applications appear in [24], where an algebra on the equivalence classes of direction objects is defined. The algebra unifies different methods for directonal qualitative reasoning that have appeared in previous work[4]. These issues are outside the scope of this paper which focusses on defining the basic concepts and implications of treating directions as objects.

1.3 Scope and Outline of the Paper

The organization of this paper is as follows: In section 2 , we define a mathematical framework and propose new ADTs for directions and orientations. A new spatial data type hierarchy is also proposed in this section. In section 3, an oriented spatial object is defined using the orientation object, and its application in spatial queries is discussed. Open geometry and its applications are illustrated in section4. Finally, the paper concludes with discussions and recommendations for future work.

2 Basic Concepts

2.1 Points, Vectors, Angles

The basic concepts we use here are points, vectors and angles, as illustrated in figure 1. A point has a position in space, which can be described using coordinates in a coordinate system. The only characteristic that distinguishes one point from another is its position. A vector, in contrast, has both

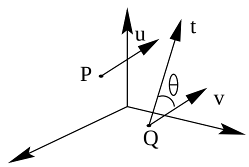


Figure 1: Diagram of points, vectors and angles

magnitude and direction but no fixed position in space. An angle between two vectors represents the direction deviation from one vector to the other. Here, by angle, we mean the smaller one between two vectors, like θ in diagram 1. In this paper, we denote points by capitalized letters, such as P, and denote vectors as lower case letters with an arrow above, such as $\vec{v}$.

For simplicity we will use the Cartesian coordinate system to represent points and vectors in this paper. The Cartesian coordinate system in 3D is a right-handed rectangular coordinate system, with three axes x, y and z perpendicular to each other and intersecting at the origin. A point P in space is represented as an ordered triple (x, y, z), where x, y and z are rectangular components of P on the x, y and z axes , respectively. Let $\vec{U_x}, \vec{U_y}$ and $\vec{U_z}$ represent three unit vectors which are the basis vectors of the coordinate system. A vector $\vec{v}$ can be defined as a linear combination of these basis vectors in the form of: $\vec{v} = a\vec{U_x} + b\vec{U_y} + c\vec{U_z}$, where a, b and c are some numbers.

Operands	Operations	Definition
Point(s)	-	$\vec{pq} = Q - P = (x_q - x_p)\vec{U_x} + (y_q - y_p)\vec{U_y} + (z_q - z_p)\vec{U_z}$
Vector(s)	+	$\vec{a} + \vec{b} = (x_a + x_b)\vec{U_x} + (y_a + y_b)\vec{U_y} + (z_a + z_b)\vec{U_z}$
	$\odot$	$\vec{a} \odot \vec{b} = \lvert a\rvert\lvert b\rvert\ cos\theta = x_a x_b + y_a y_b + z_a z_b$
	$\times$	$\vec{a} \times \vec{b} = \lvert a\rvert\lvert b\rvert\ sin(\theta)\ \vec{u}$, where $\vec{u}$ is a unit vector, perpendicular to $\vec{a}$ and $\vec{b}$
	scale	$m\vec{a} = mx_a\vec{U_x} + my_a\vec{U_y} + mz_a\vec{U_z}$, where m is a number
Point, Vector	+	$R = P + \vec{a} = R(x_p + x_a, y_p + y_a, z_p + z_a)$, a point away from P with distance $\lvert\vec{a}\rvert$ along the direction of $\vec{a}$

Table 1: Operations on vectors and points

Table 1 summarizes representative operations on points and vectors. Here $P(x_p, y_p, z_p)$ and $Q(x_q, y_q, z_q)$ are two points, θ is the angle between vectors $\vec{a}$ and $\vec{b}$, where $\vec{a} = x_a\vec{U_x} + y_a\vec{U_y} + z_a\vec{U_z}$ and $\vec{b} = x_b\vec{U_x} + y_b\vec{U_y} + z_b\vec{U_z}$. We can see that the subtraction operation(-) is applied to point pairs, resulting in a vector. The operations of addition(+), subtraction(-), scale multiplication, dot product($\odot$), and cross product($\times$) are available between vector operands. The definitions are given in terms of the components in the Cartesian coordinate system. There is only one operation(+) available between a point and a vector which actually produces another point. The associativity, commutativity and distribution properties of these operators will be examined in the future work towards integrating vector objects in query languages.

2.2 Direction and Orientation

Direction is defined as a unit vector, i.e., a vector with magnitude equal to 1. Table 2 defines the operations on directions.

Operations	Definition
composition	$\vec{d}1 + \vec{d}2 = \frac{\vec{d}1+\vec{d}2}{\lvert\vec{d}1+\vec{d}2\rvert}$
deviation	$cos\theta = \vec{d_1} \odot \vec{d_2}$
reverse	$(-1) \times \vec{d}1$
between[1]	$\vec{d}$ between $\vec{d_1}$ and $\vec{d_2}$ if $\exists c_1 > 0, c_2 > 0$ s.t. $\vec{d} = c_1\vec{d_1} + c_2\vec{d_2}$
among[2]	$\vec{d}$ among $\vec{d_1}$, $\vec{d_2}$ and $\vec{d_3}$ if $\exists c_1 > 0, c_2 > 0, c_3 > 0$ s.t. $\vec{d} = c_1\vec{d_1} + c_2\vec{d_2} + c_3\vec{d_3}$

Table 2: Operations on Directions

The operations on directions can be classified into three categories. The first category contains the operations that produce new directions. *Composition* and *reverse* are operations in this category. The *composition* operation is actually achieved by *vector-addition*, and the resulting vector is scaled down by its magnitude to be a unit vector, which represents the new direction. The *reverse* operator produces the reverse direction vector. The second category of the operations is to calculate the deviation between two directions. Operator *deviation* calculates the cosine of the angle between two directions, and hence gives the deviation of one direction from the other. A pair of vectors are orthogonal if their dot-product returns zero, i.e., they have 90^0 deviation. The last category of the operations is to test the relationships among directions. The operators *between* and *among* belong to this category. In figure 2, $\vec{d}$ is between $\vec{d_1}$ and $\vec{d_2}$; however, $\vec{d_1}$ is not between $\vec{d}$ and $\vec{d_2}$. As we will see in later sections, these three categories

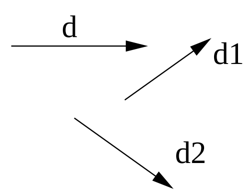


Figure 2: between operator

of direction operations make the modeling of direction concise and flexible.

Orientation is modeled as a spatial object which consists of a point of origin and N pair-wise orthogonal directions, where N is the dimension of the embedding space. The origin point and the N directions form a Cartesian coordinate system. In 3D space, the three directions may be labeled the Back-Front, Left-Right, and Below-Above directions of the orientation. Formally, we can define orientation and its operations in 3D space as follows:

[1] This definition works well as long as vector $\vec{d_1}$ is not parallel to vector $\vec{d_2}$. The parallel case can be handled in a user defined manner.

[2] Here $\vec{d_1}, \vec{d_2}$, and $\vec{d_3}$ are not in the same plane.

Orientation is a quadruple O= $\langle$OP, $\vec{front}$, $\vec{right}$, $\vec{above}\rangle$, where OP is a point, and $\vec{front}$, $\vec{right}$, $\vec{above}$ are three orthogonal directions. It has two operations:

- $translate(O, \vec{v}) = Orientation\ O' = \langle translate(OP, \vec{v}),\ \vec{front},\ \vec{right},\ \vec{above}\rangle$;
- $rotate(O, rotationMatrix) = Orientation\ O' = \langle OP, \vec{front}_n, \vec{right}_n, \vec{above}_n\rangle$, where $(\vec{front}_n, \vec{right}_n, \vec{above}_n)$= $(\vec{front}, \vec{right}, \vec{above})\odot\ rotationMatrix$;

An example of the rotation Matrix which rotates the orientation along the $\vec{above}$ axis for an angle θ is[19]:

$$R^{\theta}_{\vec{above}} = \begin{bmatrix} cos(\theta) & -sin(\theta) & 0 \\ sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Table 3 gives an illustration of these operations.

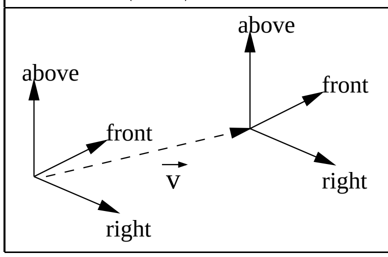
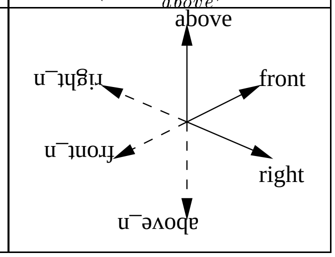
translate(O, $\vec{v}$)	rotate(O, $R^{180^0}_{\vec{above}}$)
	

Table 3: Operations on orientation

Abstract Data Types for Modeling Direction

As a summary of the above discussion, the definition of ADTs for vector, direction and orientation is given in Table 4. The C++ like syntax is used here. The column labeled *attributes* declares the member variables of each class, and the *representative operations* column declares the interfaces of operations for each class. The vector class has three member variables of the real type. These variables represent the three coefficients when the vector is written as the linear combination of basis vectors of the coordinate system. The constructor constructs a vector object, given three real arguments. The five vector operators are declared as member operator functions. Direction is defined as a subclass of the vector class. It can be constructed either from a vector or from an ordered triple. Besides inheriting attributes and operators from the vector class, direction also adds new operators such as *between* and *among*. The orientation class has four member variables which form a Cartesian coordinate system. *Translate* and *rotate* are declared as the member operator functions of the orientation class.

2.3 Frame of Reference: Absolute, Object/Viewer-based Directions

In this section, we will deal with point-based objects for simplicity. There are three different perspective systems. We define absolute directions(e.g. north, south), object-based directions(e.g. left, above), and

ADT	attributes	representative operations

vector	x-comp: float;
	y-comp: float;
	z-comp: float;
		vector(float, float, float); // constructor
		float magnitude();
		vector operator + (vector);
		vector operator scale();
		float operator dot-product (vector);
		vector operator × (vector);

Direction
subclass of vector
	inherited from vector
	constraint = unit magnitude
		direction(float, float, float); //constructor
		direction(vector); // constructor
		direction operator Composition(direction);
		direction operator reverse();
		float operator deviation(direction);
		boolean between(direction, direction);
		boolean among(direction, direction, direction);

Orientation	$OP : point$;
	$\vec{front} : direction$;
	$\vec{right} : direction$;
	$\vec{above} : direction$;
		orientation(float, direction, direction,direction);
		// constructor
		orientation operator translate(vector);
		orientation operator rotate(rotate-matrix);

Table 4: Abstract Data Types for direction and orientation

viewer-based directions.

Absolute Direction

Absolute direction in embedding space is defined as a relationship among objects based on their locations in embedding space. For simplicity here, we will ignore the elevation and earth's curvature and use a projection-based model to map the space to a 2D local coordinate system with north up and east right[1].

We model absolute directions in two ways: constant direction objects and directional predicates. Constant direction objects are created for frequently asked directions, e.g: $\vec{east}$, $\vec{west}$, $\vec{north}$, $\vec{south}$, $\vec{NW}$, $\vec{NE}$,$\vec{SE}$,$\vec{SW}$. Table 5 illustrates how we can define the constant directions in terms of coordinates in local embedding space. The unit vector is denoted by an ordered pair(a, b) which represents vector $a\vec{U_{east}} + b\vec{U_{north}}$.

unit vector	(1,0)	(0,1)	(-1,0)	(0,-1)	$(\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}})$	$(-\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}})$	$(-\frac{1}{\sqrt{2}}, -\frac{1}{\sqrt{2}})$	$(\frac{1}{\sqrt{2}}, -\frac{1}{\sqrt{2}})$
directions	$\vec{east}$	$\vec{north}$	$\vec{west}$	$\vec{south}$	$\vec{NE}$	$\vec{NW}$	$\vec{SW}$	$\vec{SE}$

Table 5: constant absolute direction

Introducing constant direction objects provides much flexibility in describing and deciding any direction. By using the operator *deviation*, we can calculate how much deviation is from one direction to another direction. We can also produce a new direction, given the deviation from a specific constant direction. This is very useful in the layout of facilities.

Second, we also provide corresponding directional predicates to make it convenient for users to

[1]For a global view, we can extend to the ellipsoidal coordinate system[13]

specify the direction relation between two objects. They are $East$, $West$, $South$, $North$, $Northwest$, $NorthEast$, $SouthWest$, $SouthEast$. For any two objects, assuming their centroids are P_1, P_2, we represent the relationship "P_1 *is to the east of* P_2" by predicate $East(P_1, P_2)$. The predicate is *true* if and only if equation $\frac{\vec{east} \odot \vec{P_2P_1}}{|\vec{P_2P_1}|} = 1$ holds. Other predicates can be defined similarly.

The above constant directions and predicates give precise directional calculation, that is, only one angle matches each case. Sometimes users may be interested only in approximate direction, such as a direction range: *between north and northwest*. The operator *between* makes this approximate directional predicate possible. If we want to test whether P_1 is *between north and northwest* of P_2, we can use the *between* operator in C++ notation as: $\vec{P_2P_1}.between(\vec{north}, \vec{NW})$.

Object-based direction

Object-based direction is the direction of the target object with respect to the orientation of the reference object. The reference object is an oriented object, while the target object may or may not have orientation. In figure 3, the person and the desk are oriented objects; the person is behind the desk and the desk is behind the person also. We will use O_B to represent the orientation of object B, and $O_B.\vec{front}$, $O_B.\vec{right}$ and $O_B.\vec{above}$ are the three directions of B's orientation. By using the direction operator *reverse* , the directions of B's behind, left and below can be described as $-O_B.\vec{front}$, $-O_B.\vec{right}$ and $-O_B.\vec{above}$, respectively.

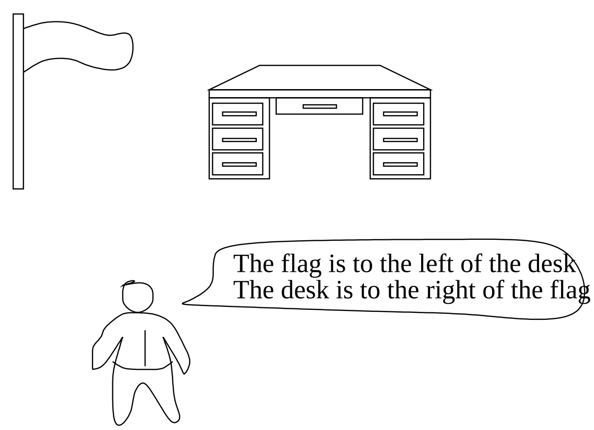


Figure 3: Viewer/Object-based direction

Given a target point object A, and a reference point object B, the direction of A relative to B can be calculated by using the deviation operation of the direction objects. First, a direction object $\vec{BA}$ is constructed from the vector $A - B$. Then the direction of point object A with respect to point object B's orientation is decided by the deviation of direction object $\vec{BA}$ from the three directions $O_B.\vec{front}$, $O_B.\vec{right}$ and $O_B.\vec{above}$. The vector dot-product is the meta operation for direction deviation operation.

Table 6 illustrates the calculation of the object-based direction. The first column consists of the directional predicates and the remaining three columns are the conditions needed to satisfy the predicates.

Direction predicates	$\vec{BA} \odot O_B.\vec{front}$	$\vec{BA} \odot O_B.\vec{right}$	$\vec{BA} \odot O_B.\vec{above}$
A in front of B = $front(A, B)$	> 0	0	0
A behind of B = $behind(A, B)$	< 0	0	0
A right to B = $right(A, B)$	0	> 0	0
A left to B = $left(A, B)$	0	< 0	0
A above B = $above(A, B)$	0	0	> 0
A below B = $below(A, B)$	0	0	< 0
A between front(B) and right(B) = $\vec{BA}.between(O_B.\vec{front}, O_B.\vec{right})$	> 0	> 0	0
$\vec{BA}.between(O_B.\vec{front}, -O_B.\vec{right})$	> 0	< 0	0
$\vec{BA}.between(-O_B.\vec{front}, -O_B.\vec{right})$	< 0	< 0	0
$\vec{BA}.between(-O_B.\vec{front}, O_B.\vec{right})$	< 0	> 0	0
$\vec{BA}.between(O_B.\vec{front}, O_B.\vec{above})$	> 0	0	> 0
$\vec{BA}.between(O_B.\vec{front}, -O_B.\vec{above})$	> 0	0	< 0
$\vec{BA}.between(-O_B.\vec{front}, O_B.\vec{above})$	< 0	0	> 0
$\vec{BA}.between(-O_B.\vec{front}, -O_B.\vec{above})$	< 0	0	< 0
$\vec{BA}.between(-O_B.\vec{right}, O_B.\vec{above})$	0	< 0	> 0
$\vec{BA}.between(-O_B.\vec{right}, -O_B.\vec{above})$	0	< 0	< 0
$\vec{BA}.between(O_B.\vec{right}, O_B.\vec{above})$	0	> 0	> 0
$\vec{BA}.between(O_B.\vec{right}, -O_B.\vec{above})$	0	> 0	< 0
A is among front(B),above(B) and left(B)= $\vec{BA}.among(O_B.\vec{front}, O_B.\vec{above}, -O_B.\vec{right})$	> 0	< 0	> 0
$\vec{BA}.among(O_B.\vec{front}, O_B.\vec{above}, O_B.\vec{right})$	> 0	> 0	> 0
$\vec{BA}.among(O_B.\vec{front}, -O_B.\vec{above}, O_B.\vec{right})$	> 0	> 0	< 0
$\vec{BA}.among(O_B.\vec{front}, -O_B.\vec{above}, -O_B.\vec{right})$	> 0	< 0	< 0
$\vec{BA}.among(-O_B.\vec{front}, O_B.\vec{above}, O_B.\vec{right})$	< 0	> 0	> 0
$\vec{BA}.among(-O_B.\vec{front}, O_B.\vec{above}, -O_B.\vec{right})$	< 0	< 0	< 0
$\vec{BA}.among(-O_B.\vec{front}, -O_B.\vec{above}, O_B.\vec{right})$	< 0	> 0	< 0
$\vec{BA}.among(-O_B.\vec{front}, -O_B.\vec{above}, -O_B.\vec{right})$	< 0	< 0	< 0

Table 6: Object-based orientation

The predicate returns *true* if and only if all three corresponding conditions hold. The first six rows illustrate the calculation of exact directions such as $front$, $left$, $above$, etc. As in row 1, if the dot-product of vector $\vec{BA}$ and direction $O_B.\vec{front}$ is greater than zero, and the dot-products between vector $\vec{BA}$ and the other two directions of orientation O_B are both zeros , then $front(A, B) = true$, which means that A is in front of B. These directional predicates are for precise direction checking, whereas the other two categories of predicates are for direction range checking. The second category of directional predicates is to test if A is located in the region between two directions. Here the operator *between* is used in C++ notation. For example, the relationship that *A is between front(A) and right(B)* is represented as predicate $\vec{BA}.between(O_B.\vec{front}, O_B.\vec{right})$, which has value *true* if and only if $\vec{BA} \odot O_B.\vec{front}$ and $\vec{BA} \odot O_B.\vec{right}$ are both greater than zero and $\vec{BA} \odot O_B.\vec{above}$ is equal to zero. The predicates in this category are for the directions in 2D space. The third category of predicates consists of eight directional predicates which test if A is among three directions in a 3D space. We represent the predicates using the operator *among*. As can be seen from the table, by using direction object $\vec{BA}$, the checking of directional predicates converts to the calculation of direction deviation which is easily performed by the dot-product operation.

Viewer-based direction

Viewer-based direction refers to the directional relationship which is measured from the viewer's perspective. In figure 3, the flag is to the left of the desk from the viewer's perspective.

There are three related components in this system: target object A, reference object B, and the viewer. The viewer has his/her own orientation, whereas objects A and B may or may not be oriented objects. Given these three components, we can estimate the direction of object A relative to object B from the viewer's perspective, in a way similar to that used with the object-based system. The similar predicates but w.r.t. the viewer are defined and are calculated by the dot-product between vector $\vec{BA}$ and the three directions of the reference orientation. The reference orientation here is the viewer's orientation O_V, and hence the three dot-products needed to be calculated are $\vec{BA} \odot O_V.\vec{front}$, $\vec{BA} \odot O_V.\vec{right}$, and $\vec{BA} \odot O_V.\vec{above}$. A similar table can be obtained as in Table 7.

Direction predicates	$\vec{BA} \odot O_V.\vec{front}$	$\vec{BA} \odot O_V.\vec{right}$	$\vec{BA} \odot O_V.\vec{above}$
$front(A, B)$	> 0	0	0
$behind(A, B)$	< 0	0	0
$right(A, B)$	0	> 0	0
$left(A, B)$	0	< 0	0
$above(A, B)$	0	0	> 0
$below(A, B)$	0	0	< 0
$\vec{BA}.between(O_V.\vec{front}, O_V.\vec{right})$	> 0	> 0	0
$\vec{BA}.among(O_V.\vec{front}, O_V.\vec{above}, O_V.\vec{right})$	> 0	> 0	> 0

Table 7: Viewer-based orientation

In the viewer-based direction system, the viewer may change his/her orientation or position(unlike objects, which are still). Using the operators *rotate* and *translate* on the orientation, we can easily generate new orientations.

Viewer-based directions are heavily used in route navigation. The route instructions can be given in terms of absolute direction, i.e., north, south, etc.. But in fact, since many people have no idea of the absolute direction, particularly in an area new to them, route instructions in absolute direction have little meaning. The directions would be much easier to understand if the instructions were given by viewer-based direction. Many web routing servers such as MapsOnUs use as many viewer-based directions as possible in their route instructions. Figure 4 is an example of a route map that starts from the Computer Science Building of the University of Minnesota to a UM apartment on the St. Paul campus. The turn-by-turn instructions are given in table 8. As can be seen from the table, viewer-based directions are used at every turn to make it easy for users to follow the descriptions intuitively.

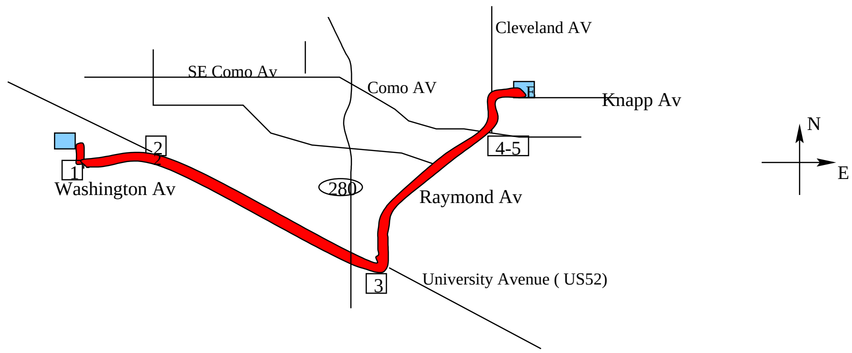


Figure 4: Route Planning

	Go	And Then	Total miles
Start		Head South on Union ST SE, From Start point(200 Union ST SE, Mpls, MN)	0.0
1	Less than .1 mi	Turn LEFT onto Washington Ave. SE	0.0
2	0.5 mi	Continue onto US52 E	0.5
3	1.4 mi	Turn LEFT onto RAYMOND Ave.	1.9
4	1.1 mi	Continue onto N Cleveland Ave	3.0
5	0.2 mi	Turn RIGHT onto Knapp street	3.2
End	Less that .1 mi	End Point(2051 Knapp Ave, St.Paul, MN)	3.2

Table 8: Turn-by-turn Directions

2.4 New Spatial Data Type Hierarchy

Some work has been done on modeling space into a spatial geometry hierarchy. The Open GIS Consortium[12] proposed a spatial object hierarchy for incorporating 2D spatial ADTs in SQL, as illustrated in figure 5.

This hierarchy consists of *Points*, *Curves*, and *Surface* and the class *GeometryCollection*. The basic operations are confined to topological and metric operations. Since these data types model only bounded shape objects, the hierarchy has limitations in modeling objects; for example, maps, rivers, roads and buildings which have their own directions/orientations. For example, with few exceptions, printed maps are all oriented with north up. This is not only for better visualization, but for more flexibility with spatial queries. It allows queries such as "Which state is to the *left* of Minnesota?" to be asked by non-geographers who may not use geographic concepts like north, west, etc.

Spatial queries such as " Is Minnesota in the north part of the USA?" can be answered by comparing the location of Minnesota with the center of the USA and its direction to the center, without the necessity of getting the information on the boundary of the USA. But in the OGIS hierarchy, spatial objects can only be modeled as bounded objects, so *north part of USA* should be represented in a polygon which needs accurate boundary information that actually has little effect on the query.

A new spatial data type hierarchy is proposed in figure 6. This hierarchy includes oriented objects and unbounded(open) objects, which could be defined using the direction and orientation objects discussed

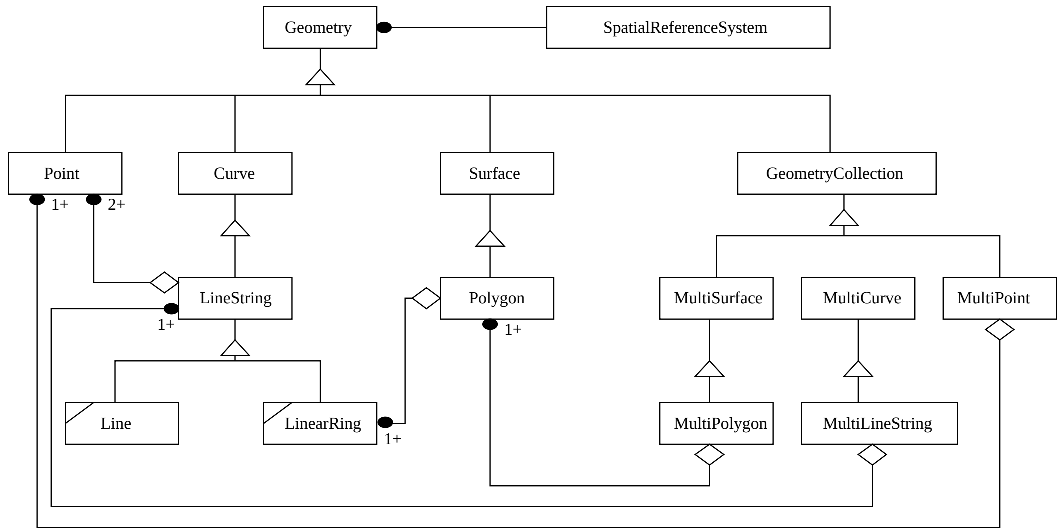


Figure 5: Spatial Data Type Hierarchy[12]

above. In the new spatial hierarchy, maps and roads can be defined as oriented objects.

New Spatial Data Type Hierarchy

Figure 6 illustrates the extended spatial data type hierarchy which consists of shape objects, vectors and oriented(directed) objects. The spatial data types and their operators can be embedded in an extended query language to implement a spatial query language.

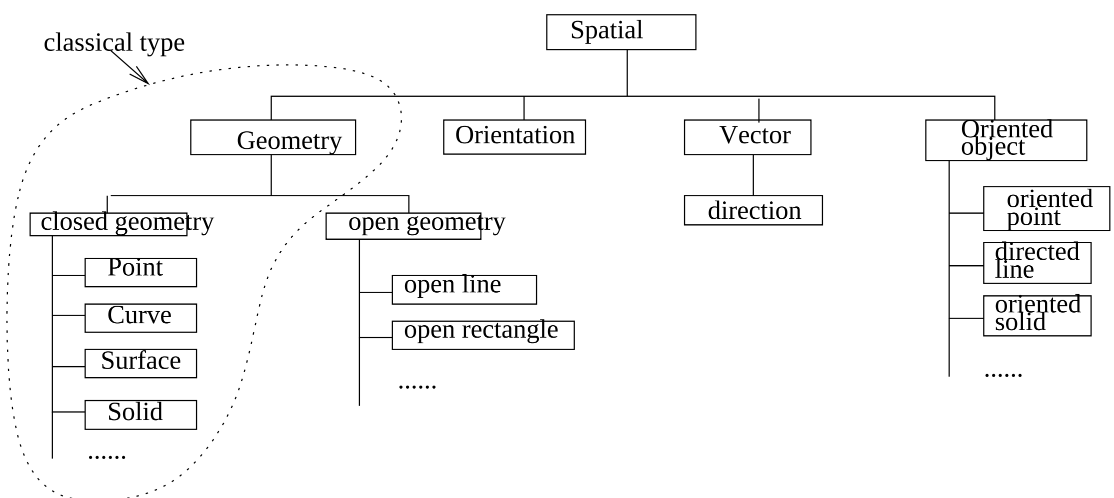


Figure 6: Extended Spatial Data Type Hierarchy

Many new operators are available in the extended spatial object model. Between vector objects and shape objects, affine transformations can be performed. The addition of these operations makes some

GIS applications easy. For example, Urban planning needs interactive placement/orientation of facilities, location-based query exploration can use affine operations, and using these operations can also improve viewer flexibility, so that we do not have to look down on earth from space. Instead, a viewer can view from any direction by performing rotation transformations.

In the later sections, the new spatial data types are discussed in detail. Their applications in geographic space are also described.

3 Defining Oriented(directed) Spatial Object

There are many objects in geographic space that have intrinsic directions/orientations in addition to their locations and shapes, such as buildings, rivers and roads. As we discussed above, modeling maps as oriented objects provides users more flexibility in query specification. It also allows users who have little knowledge of geographic space to query directions intuitively, based on their own orientations. As in the routing example given in figure 4, the turn-by-turn directions in table 8 mostly use viewer-based direction to make the information more meaningful to non-geographical users. But since there is no information on the orientation of the starting address, the first directional instruction is based on absolute direction SOUTH. If we know the orientation of the exit of the building, then we can use viewer-based direction instead, since in this case the viewer's orientation is aligned with the building exit's orientation. The instruction then may be "Turn Right onto Union Street SE", which is very easy for any user to follow.

Table 9 shows some queries that can only be answered when the reference objects and/or viewers are modeled as oriented objects.

Queries
List the swamps in front of the tank.
List the farm fields suitable for tank movement, and are left to Lake A as viewed by Viewer 1.
Let's move back a little.
How far is the next intersection down this road?

Table 9: Queries Need Orientation Information

Introducing the *Orientation* class into the spatial object hierarchy makes it easy to formalize oriented objects. In addition to the attributes such as *location* and *shape* for the geometric shape objects, an oriented object has an attribute *orientation*, which is an instance object of class orientation. The operators available for oriented objects include both topological operators and direction and orientation operators.

The ADT for oriented objects can be specified in Oracle8.0[25]. We first define the direction object type and orientation object in Oracle 8.0. In the following, we assume *Geometry* is the spatial type defined by OGIS[12]. It could be a point, a line, a polygon , and etc.

```
CREATE   TYPE DirectionType AS OBJECT (
         x NUMBER,
         y NUMBER,
         z NUMBER,
         MEMBER FUNCTION composition(aDirection IN DirectionType) RETURN DirectionType,
         MEMBER FUNCTION reverse RETURN DirectionType,
         MEMBER FUNCTION deviation(aDirection IN DirectionType) RETURN NUMBER,
         MEMBER FUNCTION between(aDirection IN DirectionType, bDirection IN DirectionType)
            RETURN NUMBER,
         PRAGMA RESTRICT_REFERENCES(default, WNDS)
);
CREATE   TYPE OrientationType AS OBJECT (
         op PointType,
         d1 DirectionType,
         d2 DirectionType,
         d3 DirectionType,
         MEMBER FUNCTION translate(aVector IN VectorType) RETURN OrientationType,
         MEMBER FUNCTION rotate(degree IN NUMBER) RETURN OrientationType,
         PRAGMA RESTRICT_REFERENCES(default, WNDS)
);
CREATE   TYPE OrientedObject AS OBJECT (
         shape Geometry,
         orientation OrientationType,
         MEMBER FUNCTION left(obj IN Geometry ) RETURN NUMBER,
         /* other functions can be defined similarily */
         PRAGMA RESTRICT_REFERENCES(default, WNDS)
);
CREATE   TYPE ViewerType AS OBJECT (
         name VARCHAR2(20),
         orientation OrientationType,
         MEMBER FUNCTION left(obj1 IN Geometry, obj2 IN Geometry) RETURN NUMBER,
         /* other functions can be defined similarily */
         PRAGMA RESTRICT_REFERENCES(default, WNDS)
);
```

We can use these data types to solve spatial queries involving oriented objects. As an example, we will show how to write the first two queries in table 9 using these data types. The first query contains object-based direction and the second one focuses on viewer-based direction. We first create a database scheme consisting of four tables, $FarmField$, $LandCover$, $Tank$, and $Viewer$. The queries are performed on this database schema.

```
CREATE TABLE FarmField(name VARCHAR2(20), type VARCHAR2(20), extent Geometry);
CREATE TABLE LandCover (name VARCHAR(20), type VARCHAR2(20), extent Geometry);
CREATE TABLE Tank (tankid VARCHAR(20), time Time, extent OrientedObject);
CREATE TABLE Viewer OF ViewerType;
```

Query 1 : List the swamps in front of the tank.

```
SELECT  L.name, L.extent
FROM    Landcover L, Tank T
WHERE   T.extent.front(L.extent) = 1 AND
        T.tankid = 'TANKID' AND L.type = 'swamps'
```

- Explanation:*front* is an object-based direction predicate with respect to the oriented object Tank. Landcover is a partition of the space based on landcover type. We search for partitions which lie in front of the tank and check their type. Example of possible types include “swamps”, “water”, “hill”.

Query 2 : List the farm fields suitable for tank movement, and are left to Lake A as viewed by Viewer1.

```
SELECT  F.name, F.extent
FROM    FarmField F, Landcover L, Viewer V
WHERE   V.left(F.extent, L.extent)=1 AND
        L.name = 'Lake A' AND V.name = 'Viewer1' AND F.type = 1
```

- Explanation: A farm field is suitable for tank movement if the crop growing inside is soft and low. The *type* field in the $FarmField$ table identify this property. $left$ is a viewer-based direction predicate with respect to the orientation of a viewer. The database will search for FarmFields that are left of the lake from Viewer1's perspective, and list all the satisfying fields suitable for tank movement.

4 Open Geometry

Open objects mean those geometries whose boundaries are partially defined, or extending beyond the data window, or infinite. To think of a query example first:

Query: Given the map of Minnesota as our data window, what's the length of the Mississippi river?

Questions: What should the GIS system answer?

(a) The length of the Mississippi river inside Minnesota.

(b) At least the length of the Mississippi river inside Minnesota, but I don't know exactly.

Since the Mississippi river extends outside Minnesota, we cannot compute the exact length of the river given only the Minnesota map, so we prefer answer (b). Modeling the river as a closed object can only return answer (a) which is not precise; however, if we model the river as an open object, then answer (b) will be obtained, which is more accurate.

4.1 Categories of Open Geometry

Table 10 categorizes the open objects based on two factors: the extent of the object and our knowledge of the object. Figure 7 gives examples of each category of open objects.

Open Object	Knowledge		Extent
	within window	beyond window	
Open-G	complete	none	finite
Open-I	complete	complete	infinite
Open-P	partial	partial	

Table 10: Defining open objects

If the object is finite and its boundary is well defined within the data window, then this is a closed object. Type Open-G refers to the object that has a finite boundary but extends beyond the data window; and the portion of the object within the data window is fully known, but the portion outside the data window is only partially known. This type of open object is useful in GIS applications. For example, if we consider the map of Minnesota as our data window, then the Mississippi river falls into the category of Open-G. Open-I refers to the object that has an infinite boundary, but we have complete knowledge of the object either within the data window or outside. Even though it seems of little interest to Geographic science since the earth has a finite surface, the category is useful in modeling direction for extended objects, as we will discuss in a later section. Moreover, Open-I and Open-G have similar

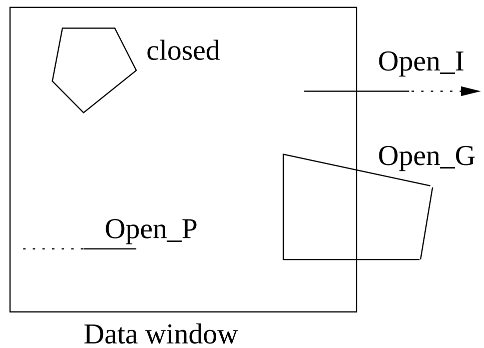


Figure 7: Examples of open shapes

topological properties. Type Open-P refers to an object of which we have only incomplete information. Dealing with this kind of object is out of the scope of our discussion. In this paper, we will focus on the category of Open-G and Open-I, and will use the term *Open object* in the places where difference between the two is unnecessary. to be differentiated.

4.2 Geometric and Topological Operations on Open Objects

In this subsection, we will examine the effects of open objects on geometric and topological operations. Table 11 and table 12 give the results for basic geometric and topological operations, respectively. Here "Closed" represents a closed object and "Open" refers to an open object. The first column of each small table contains the first argument type, and the first row contains the second argument type. For example, $Difference$(Closed, Open) = Closed, and $Difference$(Open, Closed) = Open. If the two arguments of $difference$ are Open objects, the result then could be either a closed object(If the two open objects extending to the same space) or an open object.

Intersection	Closed	Open
Closed	Closed	Closed
Open	Closed	Open/ Closed

Union	Closed	Open
Closed	closed	open
Open	Open	open

Difference	Closed	Open
Closed	Closed	Closed
Open	open	Open/ Closed

Table 11: Domains and Ranges for Geometric Operations on Open Objects

ETDOC	Closed	Open
Closed	boolean	boolean
Open	boolean	boolean/ don't know

Within	Closed	Open
Closed	boolean	boolean
Open	False	boolean/ don't know

Table 12: Domain and Ranges for Logical Operations on Open Object. **ETDOC** stands for one of $\{Equals, Touches, Disjoint, Overlap, Crosses\}$

Figure 8 illustrates the four cases of operation $Difference$ with an open shape involved. The pattern parts represent the difference between two objects. It is worth to mentioning here that since the open shapes E and F extend to the same region, so $Difference(E, F)$ is a closed shape.

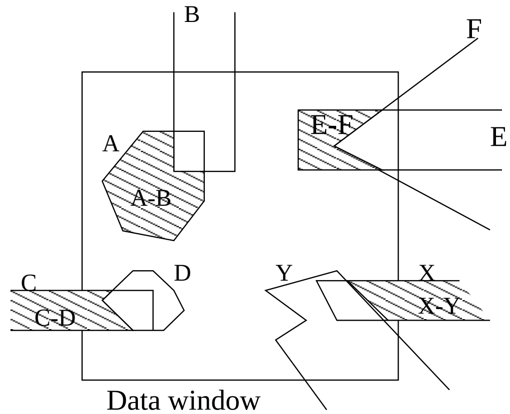


Figure 8: Operation $Difference$ on open shapes

In table 12, we combine operations that have the same results into one table. Here ETDOC represents one of the operations in the set of {*Equal, Touches, Disjoint, Overlap, Crosses*}. The result of the operations could be either *boolean* or *don't know*. *don't know* is a special value which has properties similar to the null value in a database. A useful observation is that topological operations involving closed objects always have boolean results.

4.3 Defining Open line, Open Rectangle

Examples of open spatial objects are open lines, open planes, and open spaces. A common way to define these in 3D Cartesian geometry is in terms of equations such as the parametric equations [17]: $x = x_0 + at, y = y_0 + bt, z = z_0 + ct$ for a line through the point (x_0, y_0, z_0), Ax+By+C=0 for a plane, and Ax+C=0 for a 3D space. It would be more convenient if there were corresponding object classes so that users could describe them at an abstract level and use them to describe the space. The *direction* object could be used for this purpose.

We define open lines and open rectangle regions here using directions and points. Table 13 shows how to formalize each open object with examples in figure 9.

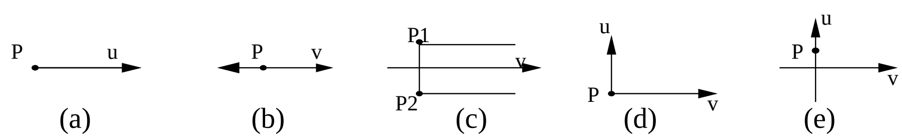


Figure 9: Examples of open shapes

There are several subtypes in each type. For an open line, it could be 1-end open(figure 9a), which needs a start point and a direction to define it, or it could be 2-end open(figure 9b), which needs an intermediate point and a direction to describe it. The open rectangle also has three subtypes: a 1-side open rectangle (9c), described by two vertices of the rectangle and one direction which is perpendicular to the edge connected by the two vertices. A 2-side open rectangle is described using two 1-end open lines which start from the vertex of the rectangle. Also a 3-side open rectangle could be defined by a

Type	Subtype	Descriptions	Examples
open line	openline1(1-end open line)	start-point, direction	(a)
	openline2(2-end open line)	intermediate-point, direction	(b)
open rectangle	openrect1:1-side open rectangle	$vertex_1$,$vertex_2$,direction, where direction perpendicular to segment formed by the two vertices	(c)
	openrect2(2-side open region)	two 1-end open lines from the only vertex	(d)
	openrect3(3-side open rectangle)	2-end open line, direction	(e)

Table 13: Defining open objects

2-end open line and a direction. It is obvious that extending to an open cube is easy.

Abstract Data Type for open lines and open rectangles

The ADTs for open objects are given in table 14 using C++ notation. The entire open object class is defined as a subclass of geometry. Geometric and topological operations available for open objects include *openUnion, openIntersection, openDifference, openWithin, openEquals, openTouches, openDisjoint, openOverlap, and OverCrosses.* The definition of the operators should be overloaded for open objects, according to table 11 and table 12. Here, we only show the attributes and constructors of each open object class, and as an illustration, we represent the spatial objects in figure 9.

ADT	attributes	constructors	examples in figure 9
Openline1	$sPoint : Point$; $dir : Direction$;	openline1(point, direction);	(a): $openline1(P, \vec{v})$
Openline2	$iPoint : Point$; $dir : Direction$;	openline2(point, direction);	(b): $openline2(P, \vec{v})$
Openrect1	$vertex_1 : Point$; $vertex_2 : Point$; $dir : Direction$;	openrect1(point, point, direction);	(c): $openrect1(P_1, P_2, \vec{v})$

Openrect2	$line_1 : Openline1$;
	
	$line_2 : Openline1$;
		openrect2(openline1, openline1);
		
		openrect2(point, direction, direction);
			(c): $openrect2(openline1(P, \vec{u}), openline1(P, \vec{v}))$
			(c): $openrect1(P, \vec{u}, \vec{v})$;

Openrect3	$line : Openline2$; $dir : Direction$;	openrect3(openline2, direction);	(d): $openrect3(openline2(P, \vec{u}), \vec{v})$

Table 14: Abstract Data Types for Open Lines and Open Rectangles

4.4 Modeling Direction Between Non-point Objects using open objects

We can use open shapes to model the directions between extended objects by converting the calculation of directional relationships to the calculation of topological relationships between objects.

Given two objects TO and RO, we want to decide the direction of target object TO related to reference object RO. Using the approach in [8], first we obtain the MBR of object RO and partition the space around object RO into nine direction tiles based on the MBR of object RO, which is a rectangle

ABCD as in figure 10. We represent each direction tile as a spatial object, as in table 15.

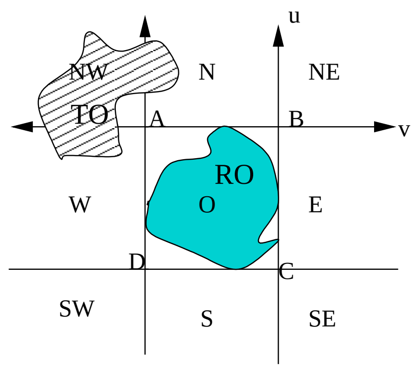


Figure 10: Object TO and RO

Rectangle	representation for figure 10
NW	$openrect2(openline2(A, \vec{u}), openline2(A, -\vec{v}))$
N	$openrect1(A, B, \vec{u})$
NE	$openrect2(openline2(B, \vec{u}), openline2(B, \vec{v}))$
E	$openrect1(B, C, \vec{v})$
SE	$openrect2(openline2(C, -\vec{u}), openline2(C, \vec{v}))$
S	$openrect1(C, D, -\vec{v})$
SW	$openrect2(openline2(D, -\vec{u}), openline2(D, -\vec{v}))$
W	$openrect1(A, D, -\vec{v})$
O	rectangle(A,B,C, D)

Table 15: Direction tile

Eight of the nine direction tiles are open rectangles. NW, NE, SE, and SW are 2-side open rectangles, and N, W, E, and S are 1-side open rectangles. In order to test the direction of object TO related to object RO, we can test into which direction tile TO falls. The calculation is then converted to the overlap relationship between object TO and the direction tiles. For example,

$$North(TO, RO) <==> openOverlap(TO, N)$$

$$NorthWest(TO, RO) <==> openOverlap(TO, NW)$$

The other directional relationship can be tested similarly. As we can see in table 12, the topological operations involving closed objects produce boolean values. The union of all the tiles that overlap with A is the region where the target object TO is located w.r.t. the object RO. For example, in figure 10, TO is North and Northwest and West of object RO.

5 Discussion

In this paper, we model direction as a new kind of spatial object using the concepts of vectors, points and angles. The basic approach here is to model direction as a unit vector. This novel object view of direction has several obvious advantages: It allows users to define the orientation of spatial objects; it also gives a richer set of predicates and operators on direction and orientation. New spatial data types such as oriented spatial objects and unbounded spatial objects can be defined as well. Finally, direction reasoning, which is useful in spatial query processing and optimization, can be implemented simply by basic vector algebra.

The benefits of the new viewpoint are obvious and promising. In future work, we would integrate vector abstract data types with an extensible query language to evaluate their efficiency in the context of various GIS applications[21, 23]. We would also like to explore the use of vectors to model values of spatial attributes such as wind-velocity, the magnetic field of the earth, etc.

Acknowledgments

This work is sponsored in part by the Army High Performance Computing Research Center under the auspices of the Department of the Army, Army Research Laboratory cooperative agreement number DAAH04-95-2-0003/contract number DAAH04-95-C-0008, the content of which does not necessarily reflect the position or the policy of the government, and no official endorsement should be inferred. This work was also supported in part by NSF grant #9631539 Thanks to to Christiane McCarthy for helping to improve the readability of the paper.

References

[1] A.Frank. Qualitative Spatial Reasoning: Cardinal Directions as an Example. *International Journal of Geographical Information Systems*, 10(3):269–290, 1996.

[2] J. Allen. Maintaining Knowledge about Temporal Intervals. *Communications of the ACM*, 26(11):832–843, 1983.

[3] D.Papadias, M. Egenhofer, and J. Sharma. Hierarchical Reasoning about Direction Relations. In *Fourth ACM Workshop on Advances in Geographic Information Systems*, pages 105–112. ACM, 1996.

[4] A. Frank. Qualitative Spatial Reasoning about Cardinal Directions. In *Auto carto 10, D.Mark and D. White, eds., Baltimore, MD*, pages 148–167, 1991.

[5] C. Freksa. Temporal Reasoning Based on Semi-Intervals. *Artificial Intelligence*, 54:199–227, 1992.

[6] C. Freksa. Using Orientation Information for Qualitative Spatial Reasoning. *Theories and Methods of Spatio-Temporal Reasoning Geographic Space*, 639:162–178, 1992.

[7] Klaus-Peter Gapp. Basic Meanings of Spatial relations: Computation and Evaluation in 3D space. *AAAI*, 2:1393–1398, 1994.

[8] R. Goyal and M. Egenhofer. The Direction-Relation Matrix: A Representation of Direction Relations for Extended Spatial Objects . In *UCGIS Annual Assembly and Summer Retreat, Bar Harbor, ME*, 1997.

[9] G.Retz-Schmidt. Various Views on Spatial Prepositions. *AI Magazine*, 9(2):95–105, 1988.

[10] John Gurney and Elizabeth Kilpple. Composing Conceptual Structure for Spoken Natural Language in a Virtual Reality Environment . 1997.

[11] R.H. Guting. An Introduction to Spatial Database Systems. *VLDB*, 3:357–399, 1994.

[12] Open GIS Consortium Inc. Opengis simple features specification for sql. http://www.opengis.org.

[13] Open GIS Consortium Inc. Spatial reference systems. http://www.opengis.org/techno/specs.htm.

[14] E. Jungert. The observer's point of view: An Extension of Symbolic Projections. *Theories and Methods of Spatial-Temporal Reasoning in Geographic Space*, 639:179–195, 1992.

[15] D. Papadias and T. Sellis. Qualitative representation of Spatial Knowledge in Two-Dimensional Space. *VLDB Journal*, 3(4):479–516, 1994.

[16] Donna J. Peuquet and Zhan Ci-Xiang. An Algorithm to Determine the Directional Relationship Between Arbitrarily-shaped Polygons in the plane. *Pattern Recognition*, 20(1):65–74, 1987.

[17] M.H. Protter and C.B. Morrey. *Modern Mathematical Analysis.* Addison-Wesley Publishing Company, Inc., 1996.

[18] P.W.Huang and Y.R. Jean. Using 2D C^+-String As Spatial Knowledge Representation For Image Database Systems. *Pattern Recognition*, 30(10):1249–1257, 1994.

[19] David F. Rogers. *Mathematical Elements for Computer Graphics*. McGraw-Hill Publishing Company, 1990.

[20] S. Shekhar, S. Ravada A.Fetterer, X.Liu, and C.T. Lu. Spatial Databases: Accomplishments and Research Needs. *University of Minnesota technical report, Csci TR97-016*, 1997.

[21] S. Shekhar, S. Ravada, V. Kumar, and etc. Parallelizing a GIS on a Shared-Address Space Architecture. *IEEE Computer*, 29(12):42–48, December 1996.

[22] Shashi Shekhar, Mark Coyle, and etc. Data Models in Geographic Information Systems. *CACM*, 40(4):103–111, 1997.

[23] Shashi Shekhar and Duen-Ren Liu. CCAM: A Connectivity-Clustered Access Method for Networks and Network Computations. *TKDE*, 9(1):102–119, 1997.

[24] Shashi shekhar, Xuan Liu, and Sanjay Chawla. Equivalence Classes of Direction Objects and Applications. Tech. Report TR99-027, University of Minnesota, Minneapolis, MN 55455.

[25] Jeffrey D. Ullman and Jennifer Widom. *A First Course in Database Systems*. Prentice Hall, 1997.

[26] Y.I.Chang and B.Y. Yang. A Prime-Number-Based Matrix Strategy for Efficient Iconic Indexing of Symbolic Pictures. *Pattern Recognition*, 30(10):1–13, October 1997.